The Algorithm of Control Program Generation for Optimization of LuNA Program Execution

Anastasia A. Tkacheva[1,2]

[1] Institute of Computational Mathematics and Mathematical Geophysics, Siberian Branch of Russian Academy of Sciences, Novosibirsk, Russia
tkacheva@ssd.sscc.ru
[2] Novosibirsk State University, Novosibirsk, Russia

Abstract. LuNA fragmented programming system is a high-level declarative system of parallel programming. Such systems have the problem of achieving an appropriate program execution performance in comparison with MPI. The reasons are a high degree of parallel program execution non-determinism and execution overhead. The paper presents an algorithm of control program generation for LuNA program. That is a step towards automatic improvement of LuNA program execution performance. Performance tests presented show effectiveness of the proposed approach.

Keywords: High performance computing, fragmented programming technology, fragmented programming system LuNA, parallel program generation

1 Introduction

Implementation of large-scale numerical models on supercomputers is difficult and to achieve good performance the programmer has to have knowledge of parallel programming. For example, the programming of the particle-in-cell method [2] requires providing dynamic load balancing, virtual layers and so on. The LuNA system [1] is being developed. Its main aim is to simplify the parallel programming process for the case of large-scale numerical models. An application program is represented in a cross-platform form with explicit parallelism. This form increases parallel program code reuse and portability, but requires complex execution algorithms in parallel programming system. It is for these reasons that there is the lack of LuNA program execution efficiency in comparison with the similar implementation using MPI [5]. On the other hand, the programmer does not have to define resources distribution. Most dynamic properties are provided automatically in LuNA system.

LuNA program can be divided into parts (subroutines). For some of the parts the most decisions on resources distribution can be made statically at compiling stage. In the paper ways to optimize LuNA program execution performance are studied. One of them is to create and use a framework for implementation of these parts. The LuNAFW framework is being developed. In LuNAFW the

application execution is based on the model of event-driven type. For using it a control program has to be developed. Inside the control program most decisions on resources distribution and operations order were partially made, and they are formulated using event handlers and calls of LuNAFW API functions for distributed or shared memory environment. The efficiency of suggested approach is presented in [5]. The manual development of control program for LuNA program is a separate time-consuming task and it does not correspond to the LuNA system development objectives. Therefore, an algorithm of control program generation is developed and proposed in the paper.

2 Related Works

There is a lack of efficiency of parallel program execution in most high-level parallel programming systems in comparison with manual implementation in MPI for the large-scale task on supercomputers. The main reason is that runtime-system can not make good decisions on resources distribution and organize parallel computation without knowledge of the application problem. To improve performance such systems usually narrow down object domain or include annotations in the language that help the run-time system to make more appropriate decisions.

For example, PaRSEC [6] system was developed for DPLASMA [7] library containing linear algebra subroutines for dense matrices. In spite of small object domain the system includes a way to define priorities of operations execution. The priorities are also used in SMP Superscalar [8]. Charm++ [9] system has the annotation language Charisma [10] to show to the run-time system which functions are related to communications. To improve performance the functional language Haskell [11] uses a coordinate language Eden [12]. The tools to create skeleton for object subdomain are also provided.

3 LuNA Fragmented Programming System

LuNA (Language for Numerical Algorithms) is a language and a parallel programming system [1] intended for implementation of large-scale numerical models on supercomputers. It is being developed in the Institute of Computational Mathematics and Mathematical Geophysics of the Siberian Branch of Russian Academy of Sciences.

In LuNA an application program is specified in a single-assignment coarse-grained explicitly parallel language LuNA as a bipartite graph of *data fragments* (DF) and *computational fragments* (CF). DFs are basically blocks of data (submatrices, array slices, etc.). CFs are applications of pure functions to DFs. A CF has a set of input DFs and a set of output DFs. Values of output DFs are computed by the CF from the values of input DFs. Such algorithm representation is called *fragmented algorithm*(FA).

LuNA program consists of a FA description in LuNA language and a dynamic load library with a set of conventional sequential procedures. CFs are implemented as calls to these procedures with input and output DFs. Execution

of all the CFs is done in accordance with partial order, that is imposed on the set of CFs by the information dependencies.

A FA is executed by the LuNA run-time system. Fragmented structure of the FA is kept in run-time, allowing the run-time system to dynamically assign CFs and DFs to different computing nodes, execute CFs in parallel (if possible), balance computational workload by redistributing CFs and DFs and so on.

The run-time system makes most decisions on FA execution dynamically. That is the reason of significant execution overhead.

The overall overhead may be divided into the following types:

- Overhead on organization of computations inside a node.
- Overhead on organization of computations among different nodes.

The previous work [13] presents ways to decrease overhead inside a node by optimizing checks for CFs being ready. This is applied for loop execution by use of Petri nets or by monolithization. It is especially important for small-grained FAs when CFs computational time is too short. The experiments in distributed computing environment show that the benefit of using those ways is limited in comparison with overhead on organization of computations among different nodes [13]. So the next aim is to optimize overhead related to distributed computing.

For optimization an approach of framework developing for an object sub-domain was chosen. The LuNAFW is such a framework based on a model of event-driven type. Unlike the existing parallel programming tools, it allows a low-level parallel programming in terms of DFs and CFs. To use it for LuNA program it is required to develop a control program where most decisions on resources distribution and CFs execution order are partially made. The decisions are represented using event handlers and framework basic API functions calls. The efficiency of the approach on the some applications is evaluated in [5].

Since the main aim of LuNA system development is to automate the process of parallel programs development, the manual control program development is not appropriate. Thus, the algorithm of control program generation was developed.

4 The Algorithm of Control Program Generation for LuNA Program

The algorithm of control program generation transforms a LuNA program into an event-driven control program for LuNAFW framework for implementation in shared and distributed memory. In a LuNA program each DF and CF has to be identified by unique (program-wide) identifiers. An identifier has an atomic form (a string) or an indexed form (a string with one or more integer indices). In LuNAFW framework CFs are distributed among different nodes using the same resources distribution strategy as in LuNA system [14].

Input of the algorithm is a FA. It includes the elements of the following types:

- CF description. It contains:
  - CF identifier.
  - Set of input DFs identifiers.
  - Set of output DFs identifiers.
  - Name of pure function in C/C++ language.
  - Set of identifiers of DFs which should be destroyed after CF execution is finished (optional parameter).
- Set of CF descriptions defined by a loop construction. It contains:
  - Name of the loop counter.
  - Value of the lower loop boundary (must be an integer).
  - Value of the upper loop boundary (must be an integer).
  - List of CF descriptions or sets of loop constructions.
- Set of output DFs of the FA.

The requirement for input FA is the ability of all information dependencies to be analyzed at compiling stage. In the case an identifier has the indexed form, the index must be either an integer constant, the name of the loop counter, or an expression of type the name of the loop counter plus/minus an integer constant.

The output of the algorithm is the generated control program represented as a C++ class. The LuNAFW program execution is based on model of event-driven type. The following handlers have to be defined:

- *onInit()* - the handler is called on program start.
- *onComputed(df_id)* - the handler is called after some DF is computed.
- *onReceived (df_id)* – the handler is called after some DF is received from other node.
- *onCfFinished (cf_id)* - the handler is called after some CF is finished execution.

Inside handlers the following functions (actions) supported by LuNAFW framework API can be called:

- *startCF (CF description)* - start CF execution.
- *checkCF (CF description)* - if all input DFs are available, start CF execution.
- *destroyDF(df_id)* - destroy DF with identifier *df_id*.
- *sendDF (df_id, rank)* – send DF with identifier *df_id* to node *rank*.
- *exit* – stop program execution.
- *getRank(identifier)* – get node number to which the CF is distributed.

The algorithm of control program generation can be divided into two stages:

1. Converter is to convert FA from data-flow-based to event-driven computation model.
2. Generator is to generate a control program from Converter output taking into account resources distribution strategy [14].

The Converter output includes:

- *Init* is the list of descriptions of CFs which have no input DFs.

- B is the list of descriptions of CFs which have no output DFs.
- *Out* is the list of output DFs of FA.
- Dictionary *GarbageCollection*:
  - Key is CF identifier.
  - Value is the list of identifiers of DFs which should be destroyed after CF execution is finished. If DF or CF identifier has indexed form, then the boundary use for each index is also defined.
- Dictionary DAG:
  - Key is DF identifier.
  - Value is the list of CF descriptions for which key is the input DF identifier. If DF or CF identifier has indexed form, then the boundary use for each index is also defined.

The dictionary *GarbageCollection* is created by analyzing CF descriptions from the input FA, specifically the set of identifiers of DFs which should be destroyed after CF execution is finished. The dictionary DAG is created by analyzing the set of input DFs identifiers for each CF descriptions. In case of the DF has indexed form, to have unique key in dictionary, the identifier needs transformation to common indexed form. All indices from indexed form are substituted to common index variables.

The output of Converter is the input of Generator. The Generator creates all the necessary handlers for CF execution, DFs internode transfers, and garbage collection in accordance with data dependencies. To generate the handler *onInit* the list *Init* is used. For each CF from it if CF is distributed to the current node, then the action *startCF* is called.

To generate the handler *onCfFinished (cf_id)* the dictionary *GarbageCollection* is used. If the key *cf_id* exists in *GarbageCollection*, then the action *destroyDF* with the corresponding value is called.

To generate the handlers *onComputed(df_id)* and *onReceived(df_id)* the dictionary DAG is used. If the key *df_id* exists in DAG, then:

- For both *onComputed* and *onReceived* handlers: The value for the key *df_id* is viewed, and if CF from it is distributed to the current node, the action *checkCF* is called.
- For *onComputed* handler: If CF from the value is distributed to the other node, then action *sendDF* is called. If the DF is needed for execution of many CFs on other node, an optimization is applied, and the DF is sent only once.

Control program is considered to be finished if all CFs from list B were executed and all DFs from the set of output DFs of FA were computed. In that case action *exit* is called.

The order of CFs execution in the generated control program does not contradict to information dependences defined in the input FA.

5 Perfomance Tests

To investigate the efficiency of the proposed algorithm an explicit finite difference method (FDM) for 3D Poisson equation solution [3] was chosen as a test application.

The experiments were conducted on MVS-10P cluster of Joint Supercomputer Center of RAS (each cluster node has two Xeon E5-2690 processors with 64 Gb RAM, 16 cores per node; nodes are connected by Infiniband FDR network). GCC 5.2.0 compiler and MPICH 3.1.4 communication library were used.

Three versions of parallel program were tested: MPI, LuNA and LuNAFW. The LuNAFW version is automatically generated using the suggested algorithm of control program generation. One MPI process per core is used. The LuNA version was tested with one thread per MPI process. The goal of the test is to evaluate weak scalability, when the problem size increases with the number of processes. In ideal case the computation time should stay the same, but in reality communication overhead make effect and the time is growing.

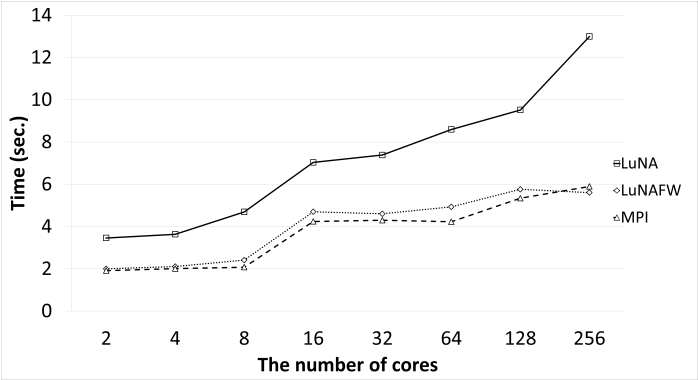

Fig. 1: Weak scalability: computation time (in sec.) dependency on the number of cores.

In Fig.1 computation times are shown for the case of a fragment size of 100x200x200 per core and for 20 iterations of FDM. The LuNAFW implementation is more efficient than LuNA version and allows achieving a good performance of parallel program in comparison with MPI implementation. The average benefit is 40%.

6 Conclusion

The ways to optimize LuNA program execution are studied. The way of using LuNAFW framework based on the model of event-driven type is chosen. To automate the control program development for LuNAFW the algorithm of its

generation is developed and considered. Performance evaluation is presented. It showed the efficiency of the suggested approach.

Acknowledgments The author would like to thank his supervisor Dr. Victor E. Malyshkin for his professional guidance and Vladislav A. Perepelkin for his constructive suggestions during the research.

References

1. Malyshkin, V.E., Perepelkin, V.A.: LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. In: PaCT 2011, LNCS, vol. 6873, pp. 53–61. Springer, Heidelberg (2011)
2. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. J. Future Generation Computer Systems, vol. 17, no. 6, pp. 755–765 (2001)
3. Kireev, S.E., Malyshkin V.E.: Fragmentation of Numerical Algorithms for Parallel Subroutines Library. J. Supercomputing, vol. 57, no. 2, pp. 161–171 (2011)
4. Kraeva, M.A., Malyshkin, V.E.: Dynamic Load Balancing Algorithms for Implementation of PIC Method on MIMD Multicomputers. J. Programmirovanie, no. 1, pp. 47–53 (in Russian) (1999)
5. Akhmed-Zaki, D.Z., Lebedev, D.V., Perepelkin, V.A.: Implementation of a three dimensional three-phase fluid flow (oilwatergas) numerical model in LuNA fragmented programming system. J. Supercomputing, vol. 73, is. 2, pp. 624-630 (2017)
6. Bosilca, G., Bouteiller, A., at all: DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In Proc. IPDPS 2011 Workshops, pp. 1151-1158 (2011)
7. Bosilca, G., Bouteiller, A., at all: Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In Proc. IPDPS 2011 Workshops, pp. 1432-1441 (2011)
8. Perez, J.M., Badia, R.M., Labarta, J.: A flexible and portable programming model for SMP and multi-cores. Technical report 03/2007, Barcelona Supercomputing Center (2007)
9. Kale L.V., Krishnan S.: CHARM++: a portable concurrent object oriented system based on C++. In Proc. of OOPSLA 93. ACM, New York, pp 91-108. (1993)
10. Chao Huang, Laxmikant, V. K.: Charisma. Orchestrating Migratable Parallel Objects. In. Proc. of the 16th Int. Symposium on High Performance Distributed Computing (HPDC), pp. 75–84 (2007)
11. Coutts, D., Loeh, A.: Deterministic parallel programming with Haskell. Comput. Sci. Eng. 14 (6), pp. 36–43 (2012)
12. Loogen, R., Ortega-Malln Y., Pea-Mar R.: Parallel Functional Programming in Eden. J. Functional Programming, No.15, is. 3, pp. 431–475 (2005)
13. Malyshkin, V.E., Perepelkin, V.A., Tkacheva A.A.: Control Flow Usage to Improve Performance of Fragmented Programs Execution. In PaCT 2015. LNCS, vol. 9251, pp. 86–90. Springer, Heidelberg (2015)
14. Malyshkin, V.E., Perepelkin V.A., Schukin, G.A.: Scalable Distributed Data Allocation in LuNA Fragmented Programming System. J. Supercomputing, vol. 73, is. 2, pp. 726-732 (2017)